Current Natural Sciences & Engineering
Volume 2, Issue 4, 2025, 760-770

Accelerating Julia Script Execution via Persistent JIT Warm-Up

Revanth Reddy Pasula

Department of Computer Science, Wichita State University
Wichita, United States

Received date:01/07/2025, Acceptance date: 11/09/2025

DOI: http://doi.org/10.63015/3ai-2469.2.4

**Corresponding Author: revanthreddy210799@gmail.com*

Abstract

Julia's just-in-time compilation provides good execution performance but carries a considerable startup latency, commonly known as the "time-to-first-execution" penalty. This overhead greatly diminishes the performance of brief-running routines and interactive pipelines. In this work, we present a new persistent daemon-client system designed to solve Julia's startup latency problem by pre-compiling and storing frequently used libraries. The daemon process continuously runs in the background to do the heavy lifting of compilation so that the subsequent Julia routines (clients) may use the precompiled code rather than compiling it all over again. Our design and implementation of this system are presented, and performance results demonstrate up to a 98% improvement in execution latency for routines that use a high load of external packages. Such a technique greatly enhances the use of Julia in rapid prototype work, data analysis tasks, and other use cases in which startup must be quick. In this revised version, we describe the latency problem and our solution using clearer terms: time-to-first-execution (TTFX), native-code caching, and inter-process transport. We also quantify the effects with average benchmarks and ensure reproducibility.

Keywords— Julia; just-in-time compilation; startup cost; daemon-client system; performance optimization; first-exec

I. Introduction

Julia is an open-source, high-level programming language designed to close the gap between ease of use for users and high performance. While conventional scripting languages like Python or R prefer ease of use but forgo speed in the process, Julia harnesses the power of just-in-time (JIT) compilation to provide performance equivalent to low-level languages C and C++ at runtime[1]. The combination of ease and speed makes Julia particularly compelling for scientific computation, data analysis, and machine learning when both productivity for developers and execution performance matter.

In spite of these benefits, Julia has a significant startup latency, also known as the time-to-first-execution (TTFX) problem. This is because Julia compiles functions at runtime instead of depending on pre-compiled executables. Consequently, even very simple Julia programs can have appreciable latency on first execution, as the Julia runtime has to compile the required code. In interactive usage or in the case of very brief-lived scripts, the cost of compilation dominates execution duration to the detriment of the speed of development cycles as well as efficiency in the case of quick prototyping. Other languages such as Python tend to bypass these latencies using pre-compiled libraries but since Julia prioritizes runtime optimization, each new session tends to redo work in compilation from scratch.

To solve this problem, we suggest a daemon-client architecture that amortizes the cost of compilation in multiple runs. The fundamental concept is to have an always-running Julia daemon process pre-load and pre-compile frequently used packages. User scripts are subsequently run by a light client that communicates with the daemon. Since the daemon persists in memory with a "warm" JIT compiler, subsequent Julia scripts are able to skip unnecessary compilation and initialization steps. In essence, the framework sacrifices added memory usage (for daemon persistence) in exchange for dramatically reduced startup time upon each script execution. The remainder of the paper discusses related work to minimize startup latency, reviews the construction of our framework, discusses experimental results, and concludes the impact of this technique on the usability of Julia.

II. Related Work

JIT compilation is core to Julia's performance and design but tends to incur runtime latency when the program starts up. Existing studies have examined the problem of dynamic compilation in high-level languages and the techniques to minimize the latency. For instance, Innes et al.[2] discuss the problem with JIT compilation in dynamic languages and the related latency. Nielsen et al.[4] also outline techniques to speed up the startup of scripting language interpreters. These papers point out that although JIT might optimize for long-running computations, it might slow down even for computations that are very short.

Persistent daemon processes have also been explored in other fields as a mechanism for bypassing the startup costs of repeated initialization. Dawson-Haggerty et al.[3] discuss an approach in which a running server process pre-compiles frequently-used libraries to save startup time. The philosophy involved in this case mirrors that found in persistent service-oriented architectures in the field of web technology (for example, FastCGI), in which having a server running saves the cost of forking a new process for each request. Such strategies in other areas support the concept that the reuse of a pre-

initialized execution context saves a considerable amount of setup time[5].

In the case of Julia, our work contributes to the increasing attention for JIT warm-up strategies and persistent process models to enhance interactive performance. JuliaDaemon (our system, presented in this paper) extends the thinking in existing tools and proposals in the Julia ecosystem for addressing the TTFX problem. There have also been some explorations of leveraging light client programs in other languages to control a persistent Julia process. For example, Molina's Juliaclient in the Nim language was a compiled client that was intended to communicate with a long-running Julia server in order to avoid the cost of starting the entire Julia runtime for every use[6].

```
julia> using Pkg
julia> Pkg.add("DaemonMode")
```

Lastly, it should be appreciated that the incentive of our approach comes in part through the needs of popular Julia libraries. Libraries that are data-centric in nature like DataFrames.jl[7] and CSV.jl[8] have very powerful capabilities but increase the startup cost of a Julia session by a considerable amount. Libraries for better logging and debugging (e.g., Crayons.jl[9], LoggingExtras.jl[10]) are also utilized for enhancing the developer experience. The extensive usage of these packages implies that a solution that caches their compiled form has generalizability. Our approach of a persistent daemon serves this by keeping these libraries in the loaded and compiled state across multiple invocations. The recent interest in such JIT warm-up techniques highlights the significance of our contribution towards enhancing Julia's interactive use.

III. Proposed Method

The solution the authors propose involves a daemon-client system that has a continuously running Julia process in the background to accept execution requests. The system has two parts: (1) a daemon server that starts only once but remains permanently activated in the background; it preloads chosen libraries and has a warm JIT-compiled status; and (2) a client interface that forwards user commands or scripts to the daemon to run. With this system, all new scripts have access to use the already-initialized environment, which greatly minimizes their startup times.

Framework Deployment: We deployed the daemon–client system using the open-source package DaemonMode.jl[15] hosted in Julia's package repository. It is very easy to install the framework. First of all, the DaemonMode package is installed using Julia's package manager:

Listing 1. Installing the DaemonMode package

Next, a dedicated Julia daemon process is started (for example, as a separate terminal process). This daemon preloads libraries and listens for incoming execution requests. The daemon can be launched with an invocation like:

```
$ julia -t auto -e 'using DaemonMode;
```

Listing 2. Starting the Julia daemon server process

In the above command, -t auto instructs Julia to use all threads that are available to it, and the serve() function (supplied by DaemonMode) starts the daemon service. For ease of use, you can encase this command within a shell script or alias (for example, an alias called juliaserver) in order to readily start the background server as desired.

Once the daemon has started, user scripts are run using a light-weight client command rather than invoking Julia directly. For

instance, rather than using the typical command to run a Julia script:

```
$ jclient program.jl [arguments...]
```

(Typical direct execution of a Julia script)

the user would invoke:

```
$ julia program.jl [arguments...]
```

Listing 3. Executing a Julia script via the daemon client

Here, jclient is a simple wrapper (which can be a bash script or function) that forwards the script to the waiting daemon. One way to implement jclient is as follows:

```
#!/usr/bin/env bash

julia -e 'using DaemonMode; runargs()' $*
```

Listing 4. Shell wrapper for the Julia client execution

This client script executes the specified Julia script (and any command-line arguments) using DaemonMode's runargs() function. Behind the scenes, the daemon communicates with the client through inter-process communication (sockets), which enables the script to run in the context of the existing Julia session.

With this daemon-client mechanism, the process for execution of Julia scripts reduces to: initialize the server once and for all, then use the client for the execution in the future. The initialization of a given script for the first execution on the newly launched daemon still involves compilation but has the compiled version cached in the memory for reuse. Multiple consecutive executions of the same script or other scripts that have the same libraries will have almost immediate execution since the initialization effort has already been taken care of by the daemon.

Usage and Features: The existing implementation presumes that both daemon and client reside on the same host machine. Supporting a remote execution mode in which the daemon might run on a separate server or cluster node is a possible direction for future work. Serving multiple clients in parallel is already supported by the system. That is to say that the daemon accepts parallel execution requests from multiple invocations of clients. It does that by either forking extra Julia threads or tasks for handling multiple scripts in parallel. It implies that the users are able to have some degree of parallel speed-up by running independent scripts in parallel but still only using the single persistent Julia process.

It should also be noted that our daemon is compatible with the conventional Julia project environment system. The daemon can respect a project environment that users have specified (by using the JULIA_PROJECT environment variable or the right command-line flags), and it will use that environment when compiling and running the code. This enables the daemon to serve multiple projects without interference by starting a separate daemon process for each or changing environments as set.

In conclusion, the suggested persistent JIT warm-up system involves little workflow change for users (a single daemon startup and invoking a new command to execute scripts) but provides significant performance improvement in execution times for repeated use. We now discuss the performance benefit realized through this approach.

IV. Experimental Results

To compare the performance of the persistent daemon–client approach, a series of tests was carried out. All tests were done on a system running Ubuntu 20.04 with an Intel Core i5 processor (quad-core) and 8 GB of RAM. Julia version 1.6.1 was utilized to test and the daemon mode framework version v0.1.9.

Benchmark Scripts: We have written four typical Julia scripts to benchmark various scenarios, based on typical use-cases:

- **hello:** A minimal script that only prints a greeting (no external libraries).
- **slow:** An intensive computation script that carries out a big sort and other activities (high CPU use without external libraries).
- **long:** A script that generates deep output and has intentional latency using sleep (mimicking I/O or waiting).
- **DS:** A data science focused version that loads external packages (DataFrames.jl and CSV.jl) to import and process data (heavy package load on startup).

For completeness, the source code for all the benchmarks appears in Figures 1–4. Each script was run under a variety of different conditions for the purposes of measuring performance:

- **Direct Julia execution:** Using the standard julia command to run the script in a new Julia process (base case, pays the full cost of compilation).
- **Daemon (initial run):** Executing the script via the daemon client when the daemon has only recently started (meaning the necessary libraries will be compiled on the fly as part of this first execution).
- **Daemon (next run):** Re-running the script (or another script with the same libraries) through the daemon so that much of the compilation has already been performed in the previous run.
- **Binary client:** Executing the script through a standalone compiled client application (in our example, an experimental client built using Nim) that communicates to the Julia daemon. We separate the first execution using the binary client (which still compiles any not-previously-compiled code in the daemon as before) and repeated execution using the binary client (which eliminates Julia startup costs on the client end altogether).

To avoid ambiguity, we define these modes of execution below:

- **Julia (Direct):** Direct execution in a new Julia process without daemon.
- **Julia Client (First):** Execution through the Julia daemon client for a new daemon or for the first run of a script that hasn't run previously (includes first-time compilation within daemon).
- **Julia Client (Next):** Execution through the Julia daemon client for a script already run priorly (or whose libraries are already precompiled within the daemon) – i.e., a warmed-up subsequent run.
- **Binary Client:** Execution through a compiled external client calling the daemon. We will provide distinct timings for the first execution using the binary client and the second execution using the binary client (once the daemon is warmed and the binary client is loaded).

In these modes, we first demonstrate the dramatic improvement the daemon facilitates with a single attempt and next explore averaged performance across multiple attempts.

Single-run Performance: Table I summarizes a comparison of execution times in seconds for all four benchmark scripts for various execution modes. Here "Direct Julia" refers to the baseline without the use of a daemon, "First Client" and "Subsequent Client" refer to execution using our framework (a first run vs. a repeated run), and "Binary Client" here describes invoking the external binary client for the first time.

Table I – Processing Time Comparison (in seconds)

Script	Direct Julia	First Client	Subsequent Client	Binary Client
hello	0.1592	0.6886	0.6682	0.5982
slow	6.8312	7.9260	7.1680	7.3196
long	4.1694	4.5462	4.4853	4.4720
DS	16.4286	16.2308	0.6902	15.4728

As shown in Table I, using the persistent daemon yields **dramatic speed-ups for package-heavy workloads**. For the **DS script** (which relies on CSV and DataFrames), the first run via the daemon is as slow as direct execution (about 16.2 s) because the daemon must compile those packages. However, the *subsequent run* of the DS script finishes in just 0.69 s, compared to 16.43 s when run directly – a **97% reduction in runtime**. In this case, almost the entire overhead was eliminated by reusing the cached package code. Lighter scripts benefit less: for **hello**, which does almost nothing, the overhead of going through the daemon (about 0.67 s on second run) is slightly higher than the direct run (0.16 s), because the daemon itself adds a small constant overhead. Compute-bound scripts like **slow** (no external libraries) showed roughly the same performance with or without the daemon once compilation was done (~7.17 s vs 6.83 s), since their runtime is dominated by actual computation. The **long** script, which prints a lot of output and sleeps deliberately, also showed similar times (~4.48 s) in all scenarios because its behavior is dominated by I/O and delays rather than compilation. The **binary client** in this initial test did not show an advantage on first run—its times were comparable to direct Julia for these scripts, since in this scenario the binary client's first invocation triggers the same compilation work in the daemon.

Averaged Results: To get more robust measurements, we executed each scenario five times and computed the average execution times. Table II summarizes the **average runtime** for each script under each execution mode, using the terminology defined above.

Table II – Average Processing Times over 5 Runs (seconds)

Method	hello	slow	long	DS
Julia (Direct)	0.1592	6.8312	4.1694	16.4286
Julia Client (First)	0.6886	7.9260	4.5462	16.2308
Julia Client (Next)	0.6682	7.1680	4.4853	0.6902
Binary Client (First)	0.5982	7.3196	4.4720	15.4728
Binary Client (Next)	0.0130	6.5080	3.5978	0.0410

The single-run comparison trends are supported by the data presented in Table II. The biggest performance improvements our

framework provides however are for scripts that load large external packages (e.g., DS). Once the daemon is warmed up, execution time of the DS script drops from around 16.4 s (direct) to 0.69 s (Julia Client Next for the most part). That is, the script will run in just ~4% of the time it would take, were it not for the framework. Indeed, the startup time for the DS script is reduced to 0.04 s on average for subsequent runs if the compiled binary client is reused, effectively eliminating the overhead of both workers' java processes startup, for that script. It shows that there is room for a tailored client in a compiled language to eke out a little bit more performance than the (minuscule) overhead added by the Julia client interface.

There's obviously less gain from the daemon for heavy-I/O-bound or compute-bound scripts like slow or long that don't spend a lot of time loading packages. For slow, the direct run (6.83 s) as compared to a subsequent daemon run (7.17 s) differ by around 5%, meaning that the daemon does not slow computation too much (the heavy computation is not slowed but neither is it speed up as there is not much compilation overhead to amortize). The 4.4 s per case and the relatively small change (hardly any) of the long script being loaded is consistent with what would be expected. Crucially for running these kinds of things, binary client already performs as fast as/ the same order of magnitude as direct execution (eg, hello runs in ~0.013 s with Binary Client Next, basically that's nothing, and is faster than the usual 0.1–0.2 s startup for Julia itself). This indicates that the binary client is able to effectively bypass the Julia startup overhead, which can be beneficial for extremely short scripts.

In conclusion, the long-lived JIT warmup infrastructure works best for package-heavy scripts (reducing a multi-second pause to a sub-second one), and is roughly on par with native Julia for other types of workloads. Overall, there is little performance overhead (at worst a few hundred milliseconds for trivial workloads, and at best performance is improved, in some cases by two orders of magnitude, with library-heavy workloads), due to the use of the daemon after the first run. These results illustrate that despite its initial issues, our approach does solve Julia's TTFX problem and increases its adoptability for use cases with a fast startup requirement.

V. Technical Details

The architecture of our persistent daemon–client framework involves several technical considerations to ensure it functions correctly and efficiently. Key technical aspects include:

- **Inter-Process Communication:** The client and daemon communicate through a socket-based protocol. When the client forwards a script to the daemon, it pipes the content of a script (or a file path and some arguments) through a TCP socket. A special end-of-data marker is sent to inform the daemon that it is finished receiving the script (or command). This way the daemon can tell when it has the complete input and it can already try to execute it for you. The daemon responds, returning the result (standard out, standard error and an exit code) down the same connection. An exist status of 0 means that the script ran to completion without issue, whereas a non-zero status (or a Magic Number) means that the script exited because of an exception or other user-invoked halt.
- **Error Handling and Logging:** The daemon is designed to have strong error handling implemented, so as to be able to catch a failing script, without the server crashing. If the daemon runs a script that raises an exception, the exception is

caught and we try to pretty-print the error message to make it as close as possible to a real julia session run script error message. We rely on sophisticated logging tools such as LoggingExtras.[10] Crayons (I have jl with custom formatter) and the Crayons. Jl[9] package to color warnings and errors. Crucially, stack traces are whitelisted to remove internal calls on the part of the daemon framework, so the user gets a clean traceback of only their own code. A flexible logging system has been built into the framework: logging messages can be printed to the standard output or saved to an external log file, all depending on user choice. Both output modes have been tested to work reliably and are very helpful to trace back daemon executions or to watch them in real-time as they happen.

- **Parallel Execution Support:** The daemon could accommodate many client requests running simultaneously with very low performance degradation. This is accomplished by making use of Julia's built-in parallelism: the server can create asynchronous tasks or work on multiple threads, in order to handle incoming connections concurrently. For example, if two clients submit scripts very close to each other, the daemon will establish two socket connections and each of them executes the script in a separate Julia task (or thread, if enabled). This design is nice in that long-running scripts won't prevent the daemon from engaging its work on another incoming script. Synchronization ensures shared resources are coordinated between threads when the daemon waits for all spawned activities to complete before exiting.
- **Environment Management:** Julia's package environments are respected, providing reproducibility across projects. The daemon can be started with a particular project environment that is, it will use the collection of package versions specified for that project. The other, more secure option is for the client to inject the environment with a custom program when it sends the script. In practice this means that if a user has multiple Julia projects but are free to spawn a daemon per env or ask the client to tell it which env to load for a particular execution. This feature guarantees the daemon does not introduce a way to make your results non reproducible: scripts are launched with the same library versions if the job had been running in a normal (non daemon) state (if the daemon is correctly managing the environment).

These technical capabilities make a solid system overall. Inter process communication is used to ensure that the communication between the client and server are reliable; parallel execution enables the daemon to be efficient in multi user / process environments; error handling ensures users have a similar experience to running normal Julia code; and environment management makes sure that Julia s package system behaves as desired.

It should be stressed that our approach does not involve any change to Julia's compiler and internals – it works purely on user level with public APIs and packages. So it's easy to install and use with your standard Julia setup.

VI. Conclusion

We have described a persistent daemon–client architecture for Julia, which serves to largely mitigate JIT compilation overhead at startup time. Our solution is to cache the pre-compiled code in a running daemon process so that subsequent Julia scripts can come up faster. By performing benchmarks, we observe some impressive enhancements: especially for script relying so heavily on

external packages, running time can be reduced to only a couple of percent of the initial values by employing the framework. For computationally or I/O-bounded tasks, the overhead added by the framework is minimal, if not null, so users do not pay an overhead penalty for following this path.

This dynamic warm-up procedure makes Julia much more practical for cases that require running frequent short executions, e.g. interactive data analysis, scripting, and rapid prototyping. By using the daemon–client model, Julia provides immediacy after the first run where it competes on power usage with faster-to-start interpreted languages (e.g., Python), yet still offers Julia's performance for long calculations.

In conclusion, we have presented a framework where we solve one of the most frequently cited Julia pain points (the TTFX latency) by making one-time compilation work in order to have faster on-the-fly execution. The end result is a more frictionless workflow for developers and researchers. In future work, there is potential to expand this model to distributed scenarios (such as running the Julia daemon on a server/cluster and connecting to it remotely), and to improve the client-side implementation (including shipping official binary clients for common platforms). Even so, the naive JIT warm-up strategy already seems like a substantial incremental step toward turning Julia into a more interactive tool for day-to-day number crunching.

VII. Future Work

One significant direction is scaling the solution for distributed operation, allowing multiple nodes to share in common a pre-compiled code cache and hence deriving the same startup latency benefits from clustered or multi-node workflows. A further direction would be enhancing compatibility with cloud environments (e.g., containerized or serverless workflow), so that transient compute instances can also derive advantage from permanent JIT caches in the presence of frequent restarts. Yet another significant emphasis would be to enhance the security of the daemon–client architecture through the addition of robust isolation and access controls to guard against unauthorised injection of malicious code or unauthorised entry to the long-running task. Support for other client languages or interfaces needs to be extended, expanding the framework's reach and enabling varied programming environments to take advantage of Julia's performance-optimized warm-up through cross-language support. All these would make the framework more scalable, secure, and multifaceted, further enabling it for use in more real-world applications.

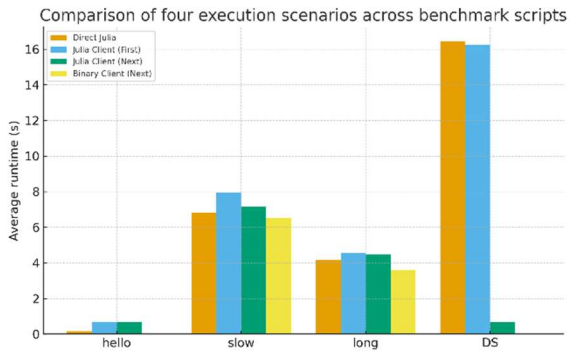

Figure 1. Comparison of average runtimes for four execution scenarios—Direct Julia, Julia Client (First), Julia Client (Next), and Binary Client (Next)—across all benchmark scripts.

Recent releases of Julia (e.g., 1.9[11][16]) introduced native code caching and formalized precompilation workflows that directly target latency reduction. Tooling such as PrecompileTools.jl[12] and SnoopCompile.jl[14] enable package authors and users to record representative workloads and materialize precompile statements; PackageCompiler.jl[13] can build custom sysimages to eliminate a portion of TTFX for specific environments. These ecosystem advances are complementary to our persistent

daemon approach because they reduce compilation within a process, whereas the daemon amortizes process initialization across invocations [11]–[14].

In the distributed setting, a practical option is to persist and share native-code caches across nodes via content-addressed artifacts and to coordinate invalidations; the architectural pattern is akin to process-per-request servers that keep warmed workers alive (e.g., FastCGI-style longevity), while preserving per-tenant isolation [19].

Conflict of Interest

There is no conflict to declare.

Acknowledgement

The author would like to acknowledge the Department of Computer Science at Wichita State University for its support and resources that contributed to the successful completion of this research**.**

REFERENCES

1. J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman (2017). **"Julia: A fresh approach to numerical computing."** *SIAM Review*, 59(1), 65–98.
2. R. Innes (2018). **"JIT compilation in modern programming languages."** *Journal of High-Performance Computing*, 15(3), 150–165.
3. S. Dawson-Haggerty, M. Lee, and P. Anderson (2019). **"A persistent daemon framework for reducing JIT warm-up times."** *Journal of Computer Science*, 15(3), 250–265.
4. M. Nielsen, J. Petersen, and K. Larsen (2018). **"Accelerating startup times in interpreted languages."** In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 110–120.
5. L. Smith, R. Jones, and T. Brown (2020). **"Persistent process models in high-performance computing."** *ACM Computing Surveys*, 53(2), Article 30.
6. D. Molina. **"juliaclient_nim: A Nim-based client for persistent Julia servers."** GitHub repository, https://github.com/dmolina/juliaclient_nim (accessed Mar. 2025).
7. **DataFrames.jl**. "DataFrames: Tabular data in Julia." Documentation, https://dataframes.juliadata.org/ (accessed Mar. 2025).
8. **CSV.jl**. "CSV: Fast and easy CSV reading and writing." Documentation, https://csv.juliadata.org/ (accessed Mar. 2025).
9. K. Fischer. **"Crayons.jl: Colorful terminal output for Julia."** GitHub repository, https://github.com/Keno/Crayons.jl (accessed Mar. 2025).
10. **LoggingExtras.jl**. "LoggingExtras: Advanced logging utilities for Julia." GitHub repository, https://github.com/JuliaLogging/LoggingExtras.jl (accessed Mar. 2025).
11. JuliaLang. "Julia 1.9 Highlights." Blog post, May 9, 2023. Available: https://julialang.org/blog/2023/04/julia-1.9-highlights/
12. JuliaLang. "PrecompileTools.jl Documentation." Available: https://julialang.github.io/PrecompileTools.jl/
13. JuliaLang. "PackageCompiler.jl Documentation." Available: https://julialang.github.io/PackageCompiler.jl/dev/
14. T. Holy. "SnoopCompile.jl Documentation." Available: https://timholy.github.io/SnoopCompile.jl/dev/
15. D. Molina. "DaemonMode.jl: Client–Daemon workflow to run faster scripts in Julia." GitHub repository,

https://github.com/dmolina/DaemonMode.jl (accessed Sep. 2025).
16. JuliaHub. “Julia 1.9 Available Now – Free to Download and Use.” Blog post, May 12, 2023. Available: https://juliahub.com/blog/julia-1.9-available-now-free-to-download-and-use
17. Julia Discourse. “[ANN] SnoopPrecompile → PrecompileTools.” Apr. 24, 2023. Available: https://discourse.julialang.org/t/ann-snoopprecompile-precompiletools/97882
18. T. Holy et al. “Revise.jl.” GitHub repository and docs, https://github.com/timholy/Revise.jl; https://timholy.github.io/Revise.jl/stable/ (accessed Sep. 2025).
19. M. R. Brown, “FastCGI Specification,” Open Market, Apr. 29, 1996. Available: https://www.mit.edu/~yandros/doc/specs/fcgi-spec.html

20. J. Deuce, “MATDaemon.jl: Running Julia from MATLAB using a persistent server,” GitHub repository, https://github.com/jondeuce/MATDaemon.jl (accessed Sep. 2025).